Record Setting Hadoop in the Cloud

By M.C. Srivas, CTO, MapR Technologies

When MapR was invited to provide Hadoop on Google Compute Engine, we ran a lot of mini tests on the virtualized hardware to figure out how to tune our software. The results of those tests were surprisingly good. The speed and consistent performance provided by Google Compute Engine even at the 99th percentile was very impressive. So we decided to run a TeraSort benchmark just to see how well we could do in a virtualized, shared environment where we didn't control the hardware, nor control the other non-MapR tenants.

An Earlier Result

We announced our first set of results on June 29, 2012 during Google's developer conference, Google I/O on a 1256 instance cluster consisting of n1-standard-4-d Google Compute Engine nodes. An n1-standard-4-d instance consists of 4 virtual cores, 1 virtual disk, and 1 virtual Ethernet interface.

The result was a fairly respectable performance of the MapR software, sorting a terabyte in 80 seconds on 1256 virtual instances. Some of the important tuning parameters for that run were:

- 4 mappers and 2 reducers on each instance
- HDFS block size = 256M
- io.sort.mb = 350M
- 2 copier threads for each reducer (mapred.reduce.parallel.copies = 2)

However, given the awesome performance of Google Compute Engine, we knew we could do much better. We focused on getting additional performance advantages out of the MapR platform.

New World Record

We are proud to announce that we were able to run the Hadoop TeraSort benchmark to **sort 1TB of data** in a world-record setting time of **54 seconds on a 1003-instance cluster** that Google graciously provided for our use. Of the 1003 instance, 998 instance ran the jobs, and 5 instance were used for control (e.g., ran the JobTracker, Zookeeper, and MapR's AdminServer). These results represent a 10% improvement on the previous world record, achieved with approximately a third the number of cores.

	MapR on GCE	Previous record	Approx. Ratio
elapsed time	54 seconds	62 seconds	
#nodes	virtual instance 1003	physical 1460	2/3
#cores	4012	11680	1/3
#disks	1003	5840	1/6
#network ports	1003	1460	2/3

This new record is also an improvement of more than 50% over our earlier result. The configuration for this run was:

- HDFS block size = 1G
- 1 mapper and 1 reducer per instance
- io.sort.mb = 1200M
- 40 copier threads per reducer

Below are the details of our work. The work will be contributed back to the community. All of the changes made for this benchmark are rolled into the next release of MapR.

To view the record-breaking TeraSort visit http://youtu.be/XbUPlbYxT8g. A more detailed video of the TeraSort is available at http://youtu.be/9iQzMoy41_k.

JobTracker Improvements

Initial analysis showed that the JobTracker (the JT) was consuming an inordinate amount of CPU, well beyond what was proportionate for the load. On an 8-core machine the JT consumed all 8 cores at 100% busy. And it wasn't due to excessive garbage collection or logging.

So we went at the JobTracker with a scalpel. We eliminated 4 or 5 superfluous threads that seemed to do nothing much except pass work items around and cause needless lock-contention. We simplified a bunch of java TreeMaps into java LinkedLists which not only decreased CPU consumption but also helped in improving the lock management. With a TreeMap, one needs a global lock on the tree, while with a LinkedList one can separate the head and tail locks and achieve much higher concurrency. With LinkedList, the locks are also held for much smaller durations during insert/remove operations. Of course, the functionality in those spots in the code didn't require a total sorted order so we could get away without a TreeMap.

There was also a massive memory copy happening inside a global lock; the memory copy happened on every map-completion event that had to be reported to the task-trackers. That piece of the code had to be re-written to circumvent the memory copy.

With those changes, the JT was no longer a serious bottleneck, and we were able to run it on a 4-core machine alongside other processes, and the cores remained fairly healthy at ~80% busy.

Task Launch SpeedUp

A major concern Hadoop users have faced for a while is that tasks are not launched until well after a job is submitted. We added instrumentation to diagnose the JobClient overheads and the TaskTracker overheads. Many shell-based operations were replaced with JNI calls, in particular fstat() and chmod/chown on the LocalFileSystem which were invoked on an average of 17 times at every task launch. The way DNS was queried was cleaned up, reducing the number of calls to DNS by a substantial number. Another major improvement was fixing the parsing of the various config files (eg, core-site.xml, mapred-site.xml, log4.properties, ...) which also happened on every task launch. All these changes resulted in much better responsiveness, and our task launch time improved by almost 20 times while reducing CPU consumption and network traffic significantly.

Reducer Scheduling

Once the JobTracker was fixed, the bottleneck shifted to the reducer itself. Each reducer fetches a list of completed map events from its TaskTracker and fetches the map output for each event. It schedules a pool of threads to fetch several outputs in parallel (mapred.reduce.parallel.copies controls the number of threads). Tracing the execution showed that the threads were not getting scheduled in time, i.e., there were long delays between when the reducer received the map events and when the fetcher thread actually tried to fetch the data. Increasing the number of threads had adverse effects, which was opposite of what we'd expected. In our earlier runs, we couldn't go beyond 6 parallel copiers per reducer without worsening the performance.

We ended up rewriting this portion of the scheduler inside the reducer. Again, we massively simplified the data structures, which enabled the locking to get fine-grained. The rewritten code could easily schedule 1000's of parallel copies very rapidly.

Network Connection Management

Bumping up the reducer copier-threads caused an unexpected problem in our system. We'd started with our earlier config of 2 reducers per instance (2000 reducers in total) with each reducer spawning 100 threads to copy. That's 200,000 simultaneous copy requests flying through the system. Each reducer was talking to a new instance for the first time, and a new socket connection had to be made for each fetch. We saw spikes of about 1000-2000 simultaneous connect requests arriving instantly at each server. But many of these connect requests would fail with ECONNREFUSED, even though the receiving server was working just fine. It had us stumped for a while. We spent quite some time debugging our RPC system to see if we were overflowing any data structures. Finally we figured out that it was the kernel dropping these TCP connect requests on the floor as its internal queues got full. Increasing the listen backlog inside the kernel eliminated the connection failures.

```
echo 3000 > /proc/sys/net/core/netdev_max_backlog
```

With the above changes, the reducers were running smoothly fetching map outputs at upwards of 100MB/s from each instance. The bottleneck shifted to the mappers.

Mapper Sorting

Internal buffering inside the MapR client software caused an interesting bottleneck. When the output for one partition is ready, the MapR mapper writes it out into an independent file (similar to what existed in Hadoop 0.15.x). But the data is not written out and accumulates in an internal buffer until either the buffer overflows or the file is closed. When the buffer overflows the data is transmitted to the server asynchronously. But if the data is small enough, the buffer will never overflow and it is the close call that finally pushes it out -- but now it is synchronous rather than asynchronous. It effectively made partition sorting proceed sequentially with the data transfer instead of in parallel. We fixed this by sorting partitions in parallel which improved the map elapsed time by 30%. (see map

Final Run

The changes gave us consistent and stable runs of the TeraSort benchmark, all of which completed under 1 minute. The runs, including the many that produced the 54-second record, were achieved with 457 fewer nodes, with approximately 1/3rd the number of cores and 1/6th the number of disks, and approximately 40% fewer network ports. Here's a report for one of the several runs that broke the record and came in at 54 seconds:

Hadoop Job job_201210181210_0063 on History Viewer

User: yufeldman
JobName: TeraSort
JobConf:
maprfs:/var/mapr/cluster/mapred/jobTracker/staging/yufeldman/.staging/job_201210181210_0063/job.xml
Job-ACLs: All users are allowed
Submitted At: 18-Oct-2012 22:16:57
Launched At: 18-Oct-2012 22:16:58 (0sec)
Finished At: 18-Oct-2012 22:17:52 (54sec)
Status: SUCCESS
Analyze This Job

Kind	Total Tasks(successful+failed+killed)	Successful tasks	Failed tasks	Killed tasks	Start Time	Finish Time
Setup	0	0	0	0		
Map	998	998	0	0	18-Oct-2012 22:16:59	18-Oct-2012 22:17:30 (30sec)
Reduce	998	998	0	0	18-Oct-2012 22:16:59	18-Oct-2012 22:17:52 (53sec)
Cleanup	0	0	0	0		

	Counter	Map	Reduce	Total
Job Counters	Aggregate execution time of mappers(ms)	0	0	21,346,738
	Launched reduce tasks	0	0	998
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Launched map tasks	0	0	998
	Data-local map tasks	0	0	998
	Aggregate execution time of reducers(ms)	0	0	46,491,572

FileSystemCounters	MAPRFS_BYTES_READ	1,000,000,079,840	1,020,017,928,072	2,020,018,007,912
	MAPRFS_BYTES_WRITTEN	1,020,021,918,260	1,000,000,008,052	2,020,021,926,312
	FILE_BYTES_READ	27,253,384	0	27,253,384
	FILE_BYTES_WRITTEN	23,425,419	23,343,583	46,769,002
Map-Reduce Framework	Map input records	10,000,000,000	0	10,000,000,000
	Reduce shuffle bytes	0	1,020,001,835,004	1,020,001,835,004
	Spilled Records	10,000,000,000	0	10,000,000,000
	Map output bytes	1,000,000,000,000	0	1,000,000,000,000
	CPU_MILLISECONDS	43,375,890	27,495,970	70,871,860
	Map input bytes	1,000,000,000,000	0	1,000,000,000,000
	Combine input records	0	0	0
	SPLIT_RAW_BYTES	79,840	0	79,840
	Reduce input records	0	10,000,000,000	10,000,000,000
	Reduce input groups	0	9,470,398,842	9,470,398,842
	Combine output records	0	0	0
	PHYSICAL_MEMORY_BYTES	1,234,250,928,128	1,455,579,328,512	2,689,830,256,640
	Reduce output records	0	10,000,000,000	10,000,000,000
	VIRTUAL_MEMORY_BYTES	2,924,949,504,000	6,105,965,477,888	9,030,914,981,888
	Map output records	10,000,000,000	0	10,000,000,000
	GC time elapsed (ms)	215,098	1,239,858	1,454,956

All of the changes that enabled the record have already been rolled into the next release of MapR. The software was installed "as is", with only a few config changes that this blog publicly discloses above (we include the job.xml file that describes the entire configuration). All the monitoring, health checks, and HA features of MapR were on, and we even increased our GUI refresh rate by 15X to see the results continuously. (the GUI displays a variety of statistics on each node, so it has to fetch about a 100 variables per node from those 1000 nodes every second). The MapR software logs over 70 variables every 10 seconds on each node for diagnostic purposes. All of this was enabled and not turned off during the performance runs.

We would like to thank Google for their continued support of this effort. It would not be possible to do this without the Google engineers providing us with a very stable and fast environment, and responding promptly to questions we had.

MapR's MapReduce Team
Yuliya Feldman, Amit Hadke, Gera Shegalov, Subhash Gopinath, Prasad Bodupalli, M. C. Srivas

Appendix

Job Configuration: JobId - job_201210181210_0063

name	value
mapred.queue.default.acl-administer-jobs	*
mapred.tasktracker.ephemeral.tasks.timeout	10000
mapred.fairscheduler.smalljob.max.maps	10
mapred.input.dir	/t.in/gen
mapred.cache.files.timestamps	1350598616866
mapred.fairscheduler.smalljob.max.reducers	10
mapred.working.dir	/user/yufeldman
mapreduce.job.submithost	m05-perfdemo.c.mapr-demo.maprtech.com.internal
mapreduce.heartbeat.100	500
mapreduce.jobtracker.recovery.maxtime	480
mapreduce.input.num.files	998
mapred.jobtracker.retiredjobs.cache.size	1000
mapred.job.shuffle.merge.percent	1.0
mapred.maxthreads.closer.threadsnumber	4
fs.s3.blockSize	33554432
keep.failed.task.files	false
mapred.output.value.class	org.apache.hadoop.io.Text
mapreduce.tasktracker.task.slowlaunch	false
mapred.map.child.java.opts	-Xmx2000m
mapred.jobtracker.restart.recover	true
mapred.cache.files.filesizes	27308
mapred.tasktracker.ephemeral.tasks.maximum	1
mapred.job.tracker.history.completed.location	/var/mapr/cluster/mapred/jobTracker/history/done
user.name	yufeldman
mapred.cache.files	maprfs:/t.in/gen/_partition.lst#_partition.lst
mapred.output.dir	maprfs:/t.out/sort
mapred.reducetask.memory.default	1500
mapreduce.job.dir	maprfs:/var/mapr/cluster/mapred/jobTracker/staging/yufeldman/.staging/job_201210181210_0063
mapred.cluster.ephemeral.tasks.memory.limit.mb	200
mapred.reduce.parallel.copies	40

fs.maprfs.impl	com.mapr.fs.MapRFileSystem
group.name	yufeldman
mapred.job.reduce.input.buffer.percent	0.75
mapred.job.name	TeraSort
mapreduce.tasktracker.reserved.physicalmemory.mb.low	0.80
mapred.tasktracker.ephemeral.tasks.ulimit	4294967296>
fs.mapr.working.dir	/user/yufeldman
mapreduce.jobtracker.staging.root.dir	/var/mapr/cluster/mapred/jobTracker/staging
fs.file.impl	org.apache.hadoop.fs.LocalFileSystem
mapred.fairscheduler.eventlog.enabled	false
mapred.job.tracker.persist.jobstatus.dir	/var/mapr/cluster/mapred/jobTracker/jobsInfo
io.sort.record.percent	0.17
mapred.reduce.tasks.speculative.execution	false
mapred.jobtracker.port	9001
mapred.jobtracker.taskScheduler	org.apache.hadoop.mapred.JobQueueTaskScheduler
mapreduce.heartbeat.10000	500
mapred.map.tasks	998
fs.default.name	maprfs:///
mapred.output.key.class	org.apache.hadoop.io.Text
mapred.jobtracker.jobhistory.lru.cache.size	5
mapred.reduce.tasks	998
mapreduce.heartbeat.10	500
mapred.maptask.memory.default	800
io.file.buffer.size	8192
mapreduce.tasktracker.jvm.idle.time	10000
fs.s3n.blockSize	33554432
mapred.tasktracker.map.tasks.maximum	1
mapred.fairscheduler.smalljob.schedule.enable	false
mapred.fairscheduler.smalljob.max.reducer.inputsize	1073741824
fs.s3.block.size	33554432
mapred.job.reuse.jvm.num.tasks	-1
mapred.jar	/var/mapr/cluster/mapred/jobTracker/staging/yufeldman/.staging/job_201210181210_0063/job.jar
mapred.tasktracker.task-controller.config.overwrite	true
mapred.tasktracker.reduce.tasks.maximum	1
mapreduce.tasktracker.prefetch.maptasks	0.0
mapred.job.shuffle.input.buffer.percent	0.7
fs.mapr.rpc.timeout	120
mapreduce.cluster.map.userlog.retain-size	-1
mapred.partitioner.class	org.apache.hadoop.examples.terasort.TeraSort$TotalOrderPartitioner
mapred.job.tracker	maprfs:///
mapred.maxthreads.generate.mapoutput	5
mapreduce.tasktracker.heapbased.memory.management	false

mapreduce.heartbeat.1000	500
fs.s3n.block.size	33554432
mapred.committer.job.setup.cleanup.needed	false
io.sort.mb	1200
mapreduce.jobtracker.node.labels.monitor.interval	120000
mapreduce.tasktracker.outofband.heartbeat	true
mapred.local.dir	/tmp/mapr-hadoop/mapred/local
mapreduce.tasktracker.group	mapr
io.sort.factor	256
mapred.create.symlink	yes
mapred.used.genericoptionsparser	true
mapreduce.maprfs.use.compression	true
webinterface.private.actions	true
mapred.fairscheduler.smalljob.max.inputsize	10737418240
hadoop.proxyuser.root.groups	root
mapreduce.task.classpath.user.precedence	false
mapred.system.dir	/var/mapr/cluster/mapred/jobTracker/system
mapred.input.format.class	org.apache.hadoop.examples.terasort.TeraInputFormat
mapreduce.cluster.reduce.userlog.retain-size	-1
mapred.map.tasks.speculative.execution	true
mapred.fairscheduler.assignmultiple	true
mapreduce.jobtracker.split.metainfo.maxsize	10000000
mapred.inmem.merge.threshold	5000000
terasort.final.sync	true
hadoop.proxyuser.root.hosts	*
mapreduce.jobtracker.recovery.dir	/var/mapr/cluster/mapred/jobTracker/recovery
mapred.task.tracker.task-controller	org.apache.hadoop.mapred.LinuxTaskController
mapred.reduce.child.java.opts	-Xmx5000m
mapred.reduce.slowstart.completed.maps	0.0
mapred.output.format.class	org.apache.hadoop.examples.terasort.TeraOutputFormat
mapreduce.job.cache.files.visibilities	true
mapreduce.reduce.input.limit	-1
mapreduce.job.submithostaddress	10.240.107.75
mapred.tasktracker.taskmemorymanager.killtask.maxRSS	false